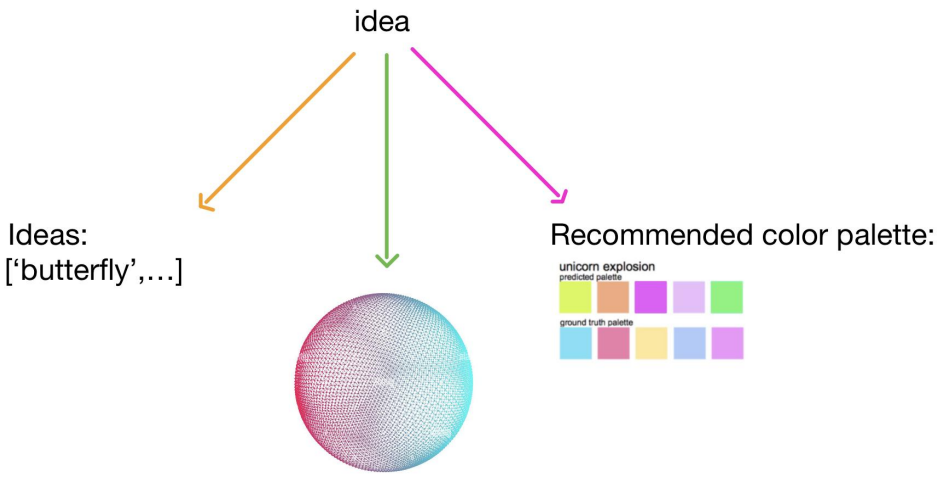
***

Cover Page

Gamifier: An Ensemble Program for the Game Design Process

By Eleanor Ye and Matthew Stephens



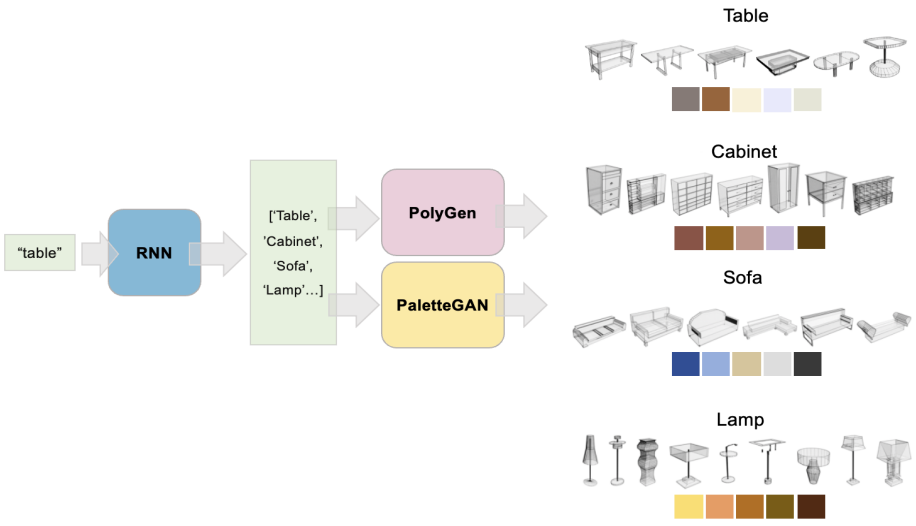
***

Introduction

The game design process has historically been a tedious one: from modelling thousands of scene contributions (either in 2D or 3D), to creatively producing unique ways to enhance the objects added to a scene. We are living in a world demanding an increasing amount of realism in the games we play. Additionally, with a correlated increase in computation power, many games of today require a higher level of complexity and intricacy within the objects used to build up the imaginative world of the game. Hence, we present Gamifier: an ensemble program consisting of three deep learning models designed to simplify the game design process. The first of the models is a RNN language model named "Thinker" — trained on both Wikipedia articles and two Harry Potter books — that takes in a user inputs (say, a single idea for an object to be added to the game) and outputs a list of ideas that are related to the first idea. Once this list has been created, it is fed into DeepMind's PolyGen model — an "Autoregressive Generative Model of 3D Meshes" — which generates 3D objects as meshes from text descriptions (DeepMind, Polygen). Additionally, the list of ideas is inputted into a Conditional Generative Adversarial Network (cGAN) — rebuilt and re-architectured following inspiration from the network "Coloring with Words: Guiding Image Colorization Through Text-based Palette Generation" of Bahng et al. (Bahng et al., Coloring with Words) — in order to output a set of color recommendations for the objects outputted by the initial ThinkerRNN model. Through combining these three models, we nicknamed the new umbrella model "Gamifier", as we believe it will simplify the game design process and unlock a range of potential applications in 3D scene creation, augmented reality, and virtual reality.

Methodology

Workflow



Preprocessing

ThinkerRNN Preprocessing

- Dataset → *Harry Potter, Prisoner of Azkaban*; *Harry Potter, Chamber of Secrets*; set of Wikipedia articles.
- Steps → Load files; split on whitespace; create vocab-dictionary mapping each word in corpus to a unique id; convert words to ids (creating an iterable set of word ids); return iterable ids and vocab-dictionary.

PolyGen Preprocessing

- Dataset → ShapeNet Core V2 dataset (ShapeNet).
- Steps → Download data; upload to google drive; retrain model on google colaboratory.

PaletteGAN Preprocessing

- Dataset → PAT dataset (9,000 multiple-word palette names, in which there are 4,312 unique words in total, and corresponding colour palettes; each palette consists of 5 colours in RGB) (Bahng et al., Coloring with Words); pretrained GloVe word embeddings (glove.840B.300d.txt and glove.6B.100d.txt, GloVe).
- Steps → Unpickle the files; pre-process the datasets into i). a customized *Dictionary* class with word-id-palette hashmaps, ii). a 4312x100 numpy array that maps word ids to 100-dimensional GloVe embeddings, and iii). a *tf.data.Dataset* with 4312 (word_id, palatte_vector) tuples.

Models

ThinkerRNN Model

The first phase of Gamifier is the Natural Language Processing Model that takes in the user input and outputs a list of related objects. The model needs to understand the context of the input in order to output an appropriate list of related objects. Hence, our ThinkerRNN is trained on a corpus consisting of two Harry Potter books, in addition to a set of Wikipedia articles, in order to learn a precise set of word embeddings holding information about the context and meaning of certain words. The model is restricted by the number of object classes PolyGen is able to produce 3D meshes for: 55 objects may be produced, however, given these objects may have different names, the model needs to understand the meaning of 129 words. Following training,

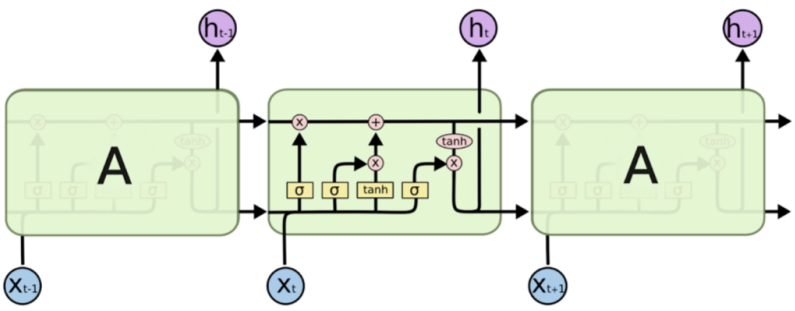
the model's word embeddings may be manipulated using cosine similarity in order to find the words most closely related to the user's input.

The architecture of the ThinkerRNN model (found in "thinker_modules") consists of learned (trainable) embeddings (shape: [vocab_size, embedding_size]) being passed into a LSTM layer (shape: rnn_size), and thereafter, the output of the LSTM being passed through a final fully-connected (or, 'Dense') layer (shape: vocab_size). The model is optimized using *tf.keras.optimizers.Adam* with a learning schedule: model training commences with an initial learning rate (0.009), but as training progresses the rate decays exponentially at a rate of 0.96 every 1000 training steps. The training corpus has been constructed using a set of wikipedia articles, in addition to two Harry Potter novels. We chose this corpus because the wikipedia articles provide an informative foundation for the model to learn the meaning of the words in the model's list of possible outputs, while the novels allow for further contextualisation to be gained from a narrative structure.

We trained the model on the corpus, performing gradient descent to calculate the loss (using *tf.keras.losses.sparse_categorical_crossentropy* loss), and then proceeded to test the model in order to calculate a perplexity score. The training function takes in the train_inputs (the preprocessed corpus), calculates the probabilities and then uses the probabilities to calculate the loss. The test function performs a feed-forward test on the model on the same data set. The test returns a perplexity score (perplexity $= e^{\frac{1}{T}\Sigma -log(P_{LM}(x^{(t+1)}|x^{(t)},\ldots,x^{(1)}))} = e^{mean\ cross-entropy\ loss\ over\ corpus}$), and once this was sufficiently low enough (<150), we considered the model trained.

ThinkerRNN (LSTM) Model Architecture

RNN-LSTM Architecture
Image source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/



PolyGen Model

At the start of this project we attempted to re-implement the PolyGen model (that uses tensorflow as its framework) in pytorch, however, as we delved deeper into the reimplementation we encountered considerable challenges translating the sonnet source code. Sonnet is a library built on tensorflow providing abstractions for tensorflow architectured models. Given that sonnet is open source, we also started translating the source code into pytorch, however, we quickly found that this would entail a deep re-implementation with significant challenges (such as solving inheritance problems and re-implementing function decorators in pytorch) that could consequently lead to an ultimate failure of the already very complex face and vertex models of PolyGen. Hence, given the amount of time and research we put into the face and vertex models of PolyGen, we made the decision to still incorporate PolyGen as a component in our project, but only as an additional feature, giving all credit to the team of researchers at DeepMind who created the wonderful model. We did, however, retrain the model ourselves based-off our test inputs, and this entailed downloading the ShapeNet Core V2 dataset (~25GB) and retraining the model on Google Colaboratory based-off our test inputs. A basic outline of the PolyGen model's architecture is described in the following paragraph.

The PolyGen model consists of two main models: a vertex model and a face model. The model takes in .obj files as input; hence, the purpose of having two models is because .obj files consist of both vertex and face descriptors. The vertex model makes use of a single decoder, while the encoder makes use of both a decoder and an encoder. In the vertex model, the number of object classes are initialized, and the maximum number of vertices are specified. The inputs are then embedded in the following order: a coordinate embedding (A) is created to track the coordinates of the learned .obj file; a positional embedding (B) is created to maintain context of embeddings; a vertical_embedding (C) represents the build in the z-dimension; and, a global embedding (D) models the build throughout the various epochs. The final embedding is the sum of the vertical, coordinate and positional embedding (embedding = C + (A + B)), and this is concatenated with a zero-embedding to achieve the correct input shape for the decoder. The final embeddings are passed into the decoder which uses *Tensor2Tensor* multi-head attention to decode the embedded inputs. The goal of the face model is to construct the mesh faces from given vertices, hence, the vertex embeddings are inputted into the face model and used in the forward pass (*face_embeddings = tf.gather(vertex_embeddings, faces_long, batch_dims=1*) to

construct the faces for the generated embeddings. This is a basic rundown of PolyGen's architecture.

PolyGen Model Architecture

(Image source: Polygen)

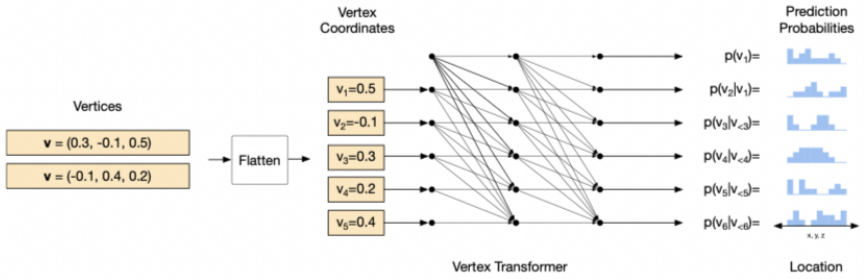


Figure 12. The vertex model is a masked Transformer decoder that takes as input a flattened sequence of vertex coordinates. The Transformer outputs discrete distributions over the individual coordinate locations, as well as the stopping token s. See Section 2.2 for a detailed description of the vertex model.

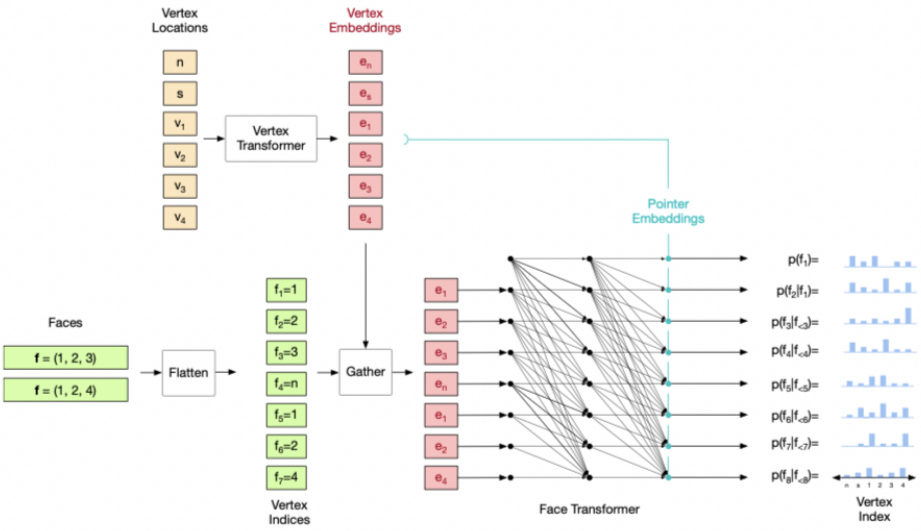
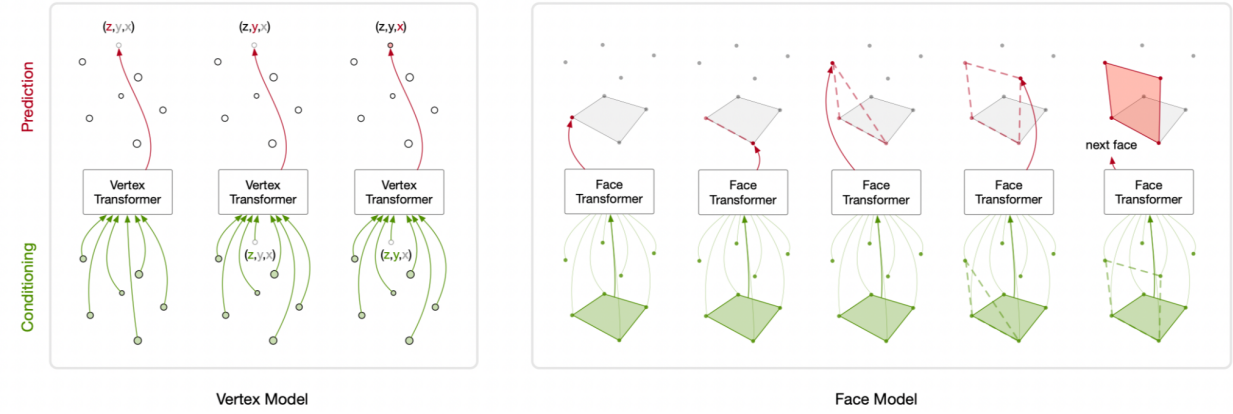


Figure 13. The face model model operates on an input set of vertices, as well as the flattened vertex indices that describe the faces. The vertices as well as the new face token n and stopping token s are first embedded using a Transformer encoder. A gather operation is then used to identify the embeddings associated with each vertex index. The index embeddings are processed with a masked Transformer decoder to output distributions over vertex indices at each step, as well as over the next-face token and the stopping token. The final layer of the Transformer outputs pointer embeddings which are compared to the vertex embeddings using a dot-product to produce the desired distributions. See Section 2.3 for a detailed description of the face model and Figure 5 in particular for a detailed depiction of the pointer network mechanism.

PolyGen Model Sampling Process

(Image source: Polygen)



PaletteGAN Model

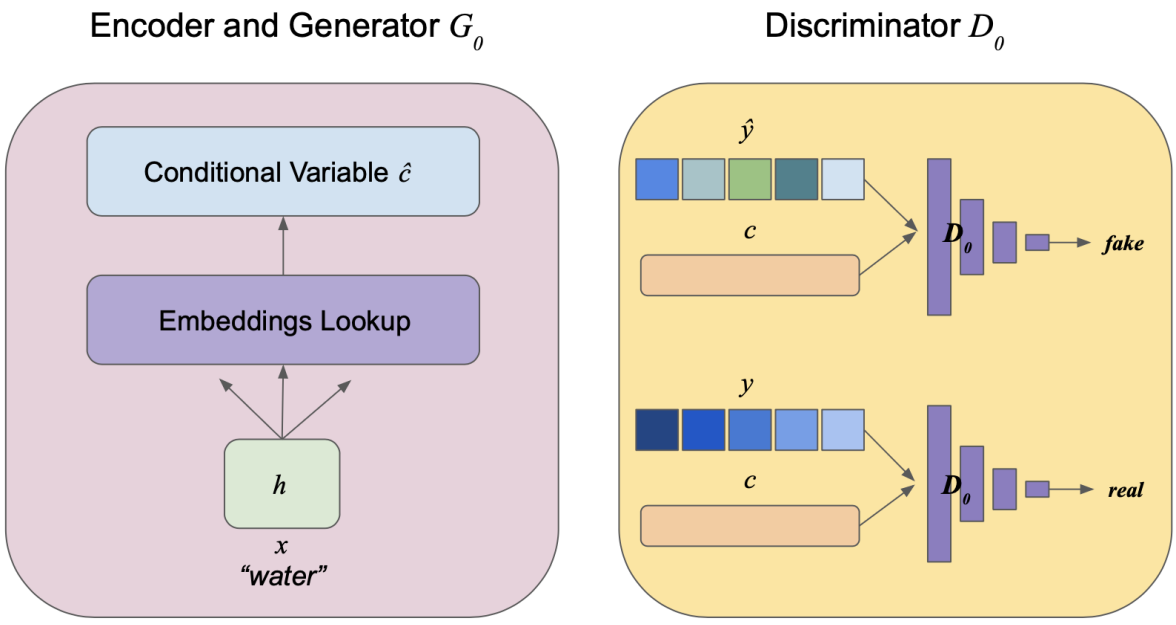
In previous sections, we discussed that the ThinkerRNN takes in an object (word) and outputs a list of relevant words, and that this list of words is then passed into PolyGen to generate a list of meshes, where one word corresponds to multiple meshes. Parallel to PolyGen, the list of words that ThinkerRNN outputs are also passed into PaletteGAN, a Conditional Generative Adversarial Network, to generate a list of 5-color color palettes, where one word corresponds to one palette. We drew our inspiration for PaletteGAN from Bahng et al.'s Text2Colors model, which is implemented in Pytorch and consists of two modules: the first one learns to generate a color palette given an input sentence, and the second one learns to use the generated color palette to colorize a given input image. Our PaletteGAN is modeled after the first module in Text2Colors, from which we kept the text-to-colors cGAN idea and data pre-processing helpers. However, we re-designed the model architecture and class structures, implemented new training and testing processes that are compatible with the overarching Gamifier workflow, and trained the model both locally and on Google Colaboratory.

The architecture of PaletteGAN (found in "paletteGAN_modules") consists of two main components - a conditional generator and a conditional discriminator, each of which is a new class inheriting from *tf.Module*. The forward pass of the Generator class takes in word ids (shape: [batch_size, 1]), looks up and returns their 100-dimensional embeddings (shape: [batch_size, 1]), passes the embeddings into "encoder" - a *tf.Sequential* - that learns to encode

the embeddings into a hidden variable (shape: [batch_size, c_dim]) by learning to compute the mu and logvar of the embeddings respectively, as well as to encode the embeddings using the mu and logvar. This hidden variable outputted by the encoder serves as the class condition for both the generator and the discriminator, hence it is called "c" in our code, and its dimension "c_dim." This variable *c* (shape: [batch_size, c_dim]) is then passed into the main generator - a *tf.Sequential* - which learns to output *y-hat*, a 5-color palette (shape: [batch_size, 15] - since colors are represented as RGB values, each color is a 1x3 vector). Then, both the color palette *y-hat* outputted by the generator, and the class condition *c* outputted by the encoder, are passed into the discriminator - again, a *tf.Sequential* - that outputs the learned classification of shape [batch_size, 1], which is the probability of each of the input palette being "real," i.e. logits.

The generator loss is the sum of three components: a vanilla generator loss calculated by taking the mean of sigmoid cross entropy between fake logits and ones, a KL loss calculated using the mu and logvar, and a huber loss calculated by taking the L1 similarity between the generated palettes and the real palettes from training dataset. The discriminator loss is computed as (the mean of sigmoid cross entropy of fake logits and zeros) + (the mean of sigmoid cross entropy of real logits and ones). In the training loop, we follow this process described above and update the gradients for the generator and discriminator using their respective losses.

PaletteGAN Model Architecture



Results

Quantitative Results

ThinkerRNN Final Perplexity Score: `50.860847`

PaletteGAN:

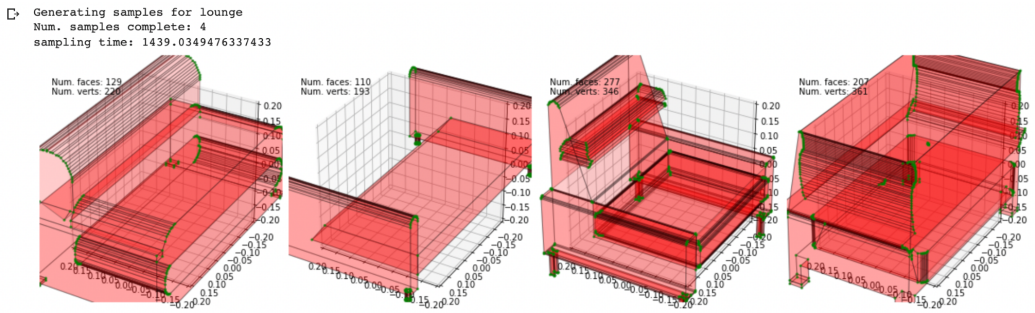
- `Epoch 0 generator loss: 92.9266 , discriminator loss: 0.010133302`
- `Epoch 500 generator loss: 50.63282 , discriminator loss: 1.6480e-05`
- `Epoch 1000 generator loss: 44.45528 , discriminator loss: 1.2741e-05`

Qualitative Results

ThinkerRNN-LSTM Output Examples	
Input	Outputs (‘Ideas’ used for object and color generation) [(‘word’, cosine similarity)], except for the input itself
lamp	`[(‘lounge’, 2.7432356), ('remote_control', 2.8682358), ('video_display', 2.9316287), ('headphone', 3.22737)]`
basket	`[('clock', 3.0484965), ('tub', 3.1929467), ('sofa', 3.3192115), ('cellular_phone', 3.4710019)]`
microwave oven	`[('washing_machine', 2.8774583), ('lamp', 3.2028174), ('flowerpot', 3.3833454), ('bowl', 4.5779705)]`
machine	`[('headphone', 2.922694), ('ash', 3.6275513), ('aeroplane', 3.823694)]`
rocket	`[('automobile', 2.934485), ('airplane', 3.2005174), ('projectile', 3.433228)]`

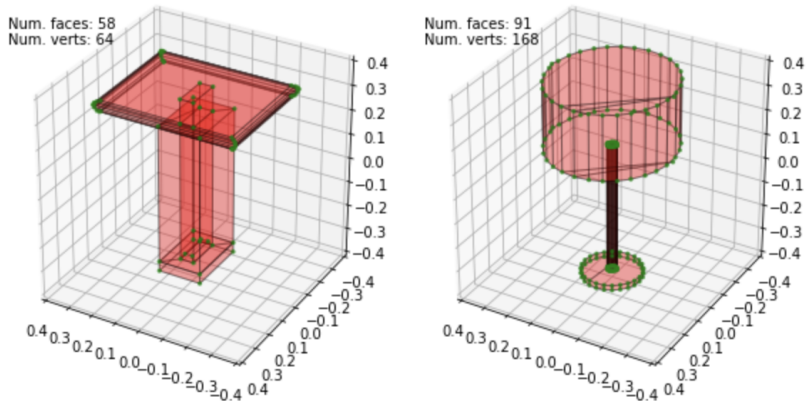
PolyGen Output:

‘Lounge’:



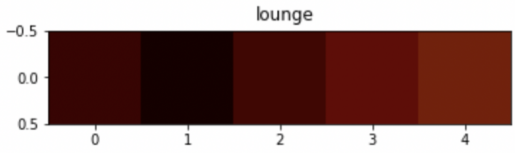
‘Lamp’:

```
Generating samples for lamp
No vertex samples completed in this batch. Try increasing max_num_vertices.
Num. samples complete: 2
sampling time: 1273.6059865951538
```



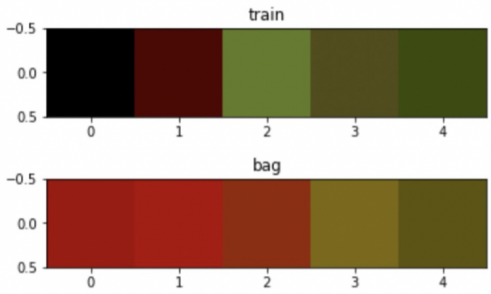
PaletteGAN Output for ‘lounge’, ‘train’, ‘bag’:

```
Fetching your colour recommendation...
    color rec: [[ 61   0   0  24   0   0  69   0   0 103   0   0 123  25   0]]
```




```
1 recs = test_run(model, ["train", "bag"])

Fetching your colour recommendation...
Loading 10183 palette names...
Making text dictionary...
train color rec: [[  0   0   0  81   1   0  97 123  35  81  76  21  57  75   0]]
bag color rec: [[165   0   0 176   0   0 150  37   0 125 105   0  94  83   0]]
```



Challenges

WorkFlow Challenges

One of the most significant challenges we faced was integrating the datasets of the three models. Our ThinkerRNN model is trained on a corpus consisting of a wide variety of words, whereas the PolyGen model can only produce a set of 55 models (129 words) and the PaletteGan model is trained on the PAT dataset consisting of 4312 unique words. Hence, the user input (and ThinkerRNN output) is largely restricted by PolyGen's dataset, and there are additional circumstances where words that are in the PolyGen dataset are not in the PAT dataset (for instance, both the RNN and PolyGen have 'remote_control', but the PAT dataset does not contain this word). As a result, we have had to do a large degree of filtering in order to ensure that the words being tested for are in all the datasets. This filtering — largely as a result of the PolyGen dataset, as it is the smallest — limits both the ThinkerRNN in addition to the PaletteGAN models. However, we performed additional tests on our PaletteGAN model with words not in the PolyGen dataset — such as on the words 'love' and 'swim' which should output 'pink-ish' and 'blue-ish' colour palettes according to the research of Bahng et al. — to ensure that the model was returning accurate colour palette predictions despite the limitations on its test dataset.

ThinkerRNN Challenges

A significant challenge we faced with the ThinkerRNN is that despite having a good perplexity score, the model is ultimately limited by the list of only 129 possible words from which it needs to output 'idea' recommendations through finding relationships between the user's input and the words in the list. The model may extract the meaning of hundreds of words from the thousands of words that pass through its various layers, however, it must then output the words most closely related to a particular word, through selecting these words from a very small, highly contrasting dataset. This is problematic because of the disparities in the dataset. For instance, the model may learn the context of the word 'aeroplane', however, in finding 'ideas' of closely related objects for the game designer, it must select from a list of words holding objects like 'trashcan', 'sofa' and 'table'. While for many of the objects this could be accounted for by taking the cosine similarity between the embeddings for each word (if the word embeddings are closely related, then the 'idea' outputs of the ThinkerRNN should be in-line with the user inputs), there were many instances where the model simply could not produce good outputs. For instance,

given the input ‘tower’, our model outputted: [('tub', 2.5892732), ('letter_box', 2.6278977), ('cellular_phone', 2.8305826), ('handgun', 2.8846939), ('pianoforte', 3.0905883), ('aeroplane', 3.5158703)], where the words are listed with increasing similarity (the number after the word represents the cosine similarity). As our main goal is to produce useful ideas — and the objects themselves — for scene contributions within game design, this might be seen as a success, rather than a failure (an ‘aeroplane’ might look great flying next to a ‘tower’ in a game), but there are cases that do simply underscore the lack of available objects to find relationships with in our dataset (for instance, while a ‘tub’ might be great in a tower, it certainly adds nothing to the particular scene).

PolyGen Challenges

As mentioned above, the task we designed for this project initially was the translation of the PolyGen model from Tensorflow to Pytorch, and more specifically the translation of the underlying Sonnet source code, which is a deep learning library build on tensorflow. After extensive efforts, we realized that this is not feasible, since PolyGen are calling numerous Sonnet functions and classes, all of which traces back to the Tensorflow version 1.36 source code with complicated functions that are beyond the scope of this class. Because of the complex nature of PolyGen, the translation process also requires advanced understanding of the Pytorch workflow. Hence, we could not successfully reimplement PolyGen in Pytorch. Instead, we switched to the task of solving a new problem, which is to combine the existing code of PolyGen with an ThinkerRNN model, “Thinker,” that we implemented from scratch, as well as a Conditional Generative Adversarial Model, named “PaletteGAN,” that we adopted from Text2Colors (Bahng et al., Coloring with Words) and rebuilt in Tensorflow, to create the Gamifier program.

PaletteGAN Challenges

One challenge with the PaletteGAN is that we were not able to directly use the pre-processed word embeddings for the training dataset from Text2Colors, since it is a .pth file that is only compatible with the Pytorch framework. We then tried to pre-process the 300-dimensional GloVe embeddings source file. However, this also failed because the file takes up 6GB space and could not be loaded in either PyCharm or Google Colab as it exceeds the RAM limits. Hence, we switched to a 100-dimensional embedding file for our training process, which would probably

cause a drop in the quality of the generated palettes as compared to the original implementation in Text2Colors.

Reflection

Initially, we set out to re-implement the PolyGen model (built on tensorflow) in pytorch, however, following a catch-22 with translating the sonnet library, we reconstructed our Gamifier model to be an umbrella model, overarching three models, including PolyGen. The results obtained from the ThinkerRNN gave us a perplexity score we aimed to achieve, and the color palettes outputted by our paletteGAN are very closely related to those described in the paper by Bahng et al., Coloring with Words. Combining these results with DeepMind's PolyGen model, we believe Gamifier succeeds at being a useful tool for game designers to use in their design process. Looking forward, as game design is something that we are both very passionate about, we hope to continue work on this project and/or aspects of this project. For instance, one idea we had at the start was to go so far as to reconstruct a scene for the game designer given a text description of the scene. Additionally, we would love to apply the PaletteGAN color palettes to the generated meshes themselves, allowing the designer to view the outputs in their full glory! As there are ways to do this, for instance, through use of programs like Blender, we believe this goal is attainable. Furthermore, as many of the project's limitations right now lie in the number of available 3D meshes for reconstruction by the PolyGen model, we have thought about re-implementing other 3D reconstruction networks, such as the work of Yufie Ye et al. in *Shelf-Supervised Mesh Prediction in the Wild*, where a neural network is trained to produce its own 3D reconstructions from image inputs. We believe this project has been a wonderful journey and we are happy to share our Gamifier with you!